

# Interprocedural Data Flow Analysis in Soot using Value Contexts

Rohan Padhye

Indian Institute of Technology Bombay  
rohanpadhye@iitb.ac.in

Uday P. Khedker

Indian Institute of Technology Bombay  
uday@cse.iitb.ac.in

## Abstract

An interprocedural analysis is precise if it is flow sensitive and fully context sensitive even in the presence of recursion. Many methods of interprocedural analysis sacrifice precision for scalability while some are precise but limited to only a certain class of problems.

Soot currently supports interprocedural analysis of Java programs using graph reachability. However, this approach is restricted to IFDS/IDE problems, and is not suitable for general data flow frameworks such as constant propagation, heap reference analysis, and precise points-to analysis with strong updates which have non-distributive flow functions.

We describe a general-purpose interprocedural analysis framework for Soot using data flow values for context-sensitivity. This framework is not restricted to problems with distributive flow functions, although the lattice must be finite. It combines the key ideas of the tabulation method of the functional approach and the technique of value-based termination of call string construction.

We instantiate our framework with a flow and context sensitive points-to analysis which enables construction of a far more precise call graph than what can be constructed in Soot currently. This is important for object oriented languages like Java where virtual method invocations cause an explosion of spurious call edges if the call graph is constructed naively. We expect any interprocedural analysis over the call graphs constructed by our framework to be more efficient and more precise.

**Categories and Subject Descriptors** F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis

**General Terms** Algorithms, Languages, Theory

**Keywords** Interprocedural analysis, context-sensitive analysis, points-to analysis, call graph

## 1. Introduction

Interprocedural data flow analysis incorporates the effects of procedure calls on the callers and callees. A context-insensitive analysis does not distinguish between distinct calls of a procedure. This causes the propagation of data flow values across interprocedurally invalid paths (i.e. paths in which calls and returns may not match)

resulting in a loss of precision. A context-sensitive analysis restricts the propagation to valid paths and hence is more precise.

Two most general methods of precise flow and context sensitive analysis are the *Functional* approach and the *Call Strings* approach [11]. The functional approach constructs summary flow functions for procedures by reducing compositions and meets of flow functions of individual statements of a procedure to a single flow function. It is then used as the flow function of the statements calling the procedure. However, constructing summary flow functions may not be possible in general. The tabulation method of the functional approach overcomes this restriction by enumerating the functions as pairs of input-output data flow values for each procedure but requires the lattice to be finite.

The call strings method remembers the calling contexts in terms of unfinished calls as call strings. However, it requires an exponentially large number of call strings. The technique of value based termination of call string construction [5] uses data flow values to restrict the combinatorial explosion of contexts and improves the efficiency significantly without any loss of precision.

Graph reachability based interprocedural analysis [9, 10] is a special case of the functional approach. Formally, it requires flow functions  $2^A \mapsto 2^A$  to distribute over the meet operation so that they can be decomposed into meets of flow functions  $A \mapsto A$ . Here  $A$  can be either a finite set  $D$  (for IFDS problems [9]) or a mapping  $D \mapsto L$  (for IDE problems [10]) from a finite set  $D$  to a lattice of values  $L$ . Intuitively,  $A$  represents a node in the graph and a function  $A \mapsto A$  decides the nature of the edge from the node representing the argument to the node representing the result. Flow function composition then reduces to a transitive closure of the edges resulting in paths in the graph.

The efficiency and precision of an interprocedural analysis is heavily affected by the precision of the call graph it uses. This is especially important for object oriented languages like Java where virtual method invocations cause an explosion of spurious call edges if the call graph is constructed naively.

Soot [12] has been a stable and popular choice for hundreds of client analyses for Java programs, though it has traditionally lacked an interprocedural framework. Bodden [4] has recently implemented support for interprocedural analysis using graph reachability. The main limitation of graph reachability is that it is not suitable for general data flow frameworks with non-distributive flow functions such as constant propagation, heap reference analysis, and precise points-to analysis with strong updates. This is because the reachability of a node  $x$  in a graph can be discovered from the reachability of its predecessors  $y$  and  $z$  independently. Thus a statement  $x = y + z$  can be processed for possibly undefined variables analysis because the required property of  $x$  can be computed from the corresponding properties of  $y$  and  $z$  independently and the effect can be combined using a meet operation. However, the same statement cannot be processed for constant propagation

because the values of  $y$  and  $z$  are required simultaneously for computing the value of  $x$  — it cannot be computed from  $y$  and  $z$  independently and the effect merged using a meet operation. The statement  $x = y + 10$  requires only the value of  $y$  and is amenable to analysis using graph reachability. Similarly, for points-to analysis, the statement  $p = q$  is amenable to analysis using graph reachability because we need the pointees of  $q$  only. However, the statement  $*p = q$  requires pointees of both  $p$  and  $q$  for precise points-to analysis with strong updates and hence is not amenable to analysis using graph reachability.

We have implemented a generic framework for performing flow and context sensitive interprocedural data flow analysis that does not require the flow functions to be distributive. It uses value-based contexts and is an adaptation of the tabulation method of the functional approach and the modified call strings method. The only restriction is that the lattice of data flow values must be finite.

Our implementation is agnostic to any analysis toolkit or intermediate representation as it is parameterized using generic types. Since our core classes are similar to the intra-procedural framework of Soot, it integrates with Soot's Jimple IR seamlessly.

We have instantiated our framework with a flow and context sensitive points-to analysis in Soot and have constructed call graphs on-the-fly using type information of the objects pointed-to by reference variables. The resulting call graphs are far more precise than those constructed by Soot's SPARK engine. Thus, any interprocedural analysis over the call graphs constructed by our framework can be expected to be more efficient and more precise.

The rest of the paper is organized as follows: Section 2 describes our method. Section 3 describes its implementation and key design decisions. Section 4 presents the results of call graph construction using our framework in Soot. Finally, Section 5 concludes the paper by describing the current status and future possibilities of our work.

## 2. Interprocedural Analysis Using Value Contexts

The tabulation method of the functional approach [11] and the modified call strings approach [5] both revolve around the same key idea: if two or more calls to a procedure  $p$  have the same the data flow value (say  $x$ ) at the entry of  $p$ , then all of them will have an identical data flow value (say  $y$ ) at the exit of  $p$ . The tabulation based method uses this idea to enumerate flow functions in terms of pairs of input-output values (eg.  $(x, y)$ ) whereas the modified call strings method uses it to partition call strings based on input values thereby reducing the number of call strings significantly. In each case a procedure body needs to be analyzed only once for an input value  $x$ .<sup>1</sup>

The two methods lead to an important conclusion: Using data flow values as contexts of analysis can avoid re-analysis of procedure bodies. We make this idea explicit by defining a *value context*  $X = \langle \text{method}, \text{entryValue} \rangle$ , where *entryValue* is the data flow value at the entry to a procedure *method*. Additionally, we define a mapping *exitValue*( $X$ ) which gives the data flow value at the exit of the procedure *method*. As data flow analysis is iterative in nature, this mapping might change over time (although it would follow a descending chain in the lattice). The new value is propagated to all callers of *method* if and when this mapping changes. With this arrangement, intraprocedural analysis can be performed for each value context independently handling flow functions in the usual way; only procedure calls need a special treatment.

The efficiency of this method arises from the observation that, in practice, the number of distinct data flow values reaching a procedure is very small. This has been corroborated by our past [5] as well as current experiments.

<sup>1</sup>In some cases in which the mapping changes, the modified call strings method can avoid re-analysis of the entire procedure.

```

1: global stack, transitions
2: procedure INITCONTEXT( $X$ )
3:   Set EXITVALUE( $X$ ) to  $\top$ 
4:   Let  $\langle \text{method}, \text{entryValue} \rangle \leftarrow X$ 
5:   for all nodes  $n$  in the body of method do
6:     Set IN( $X, n$ ) and OUT( $X, n$ ) to  $\top$ 
7:   end for
8:   Set IN( $X, \text{ENTRYNODE}(\text{method})$ ) to entryValue
9:   Initialize WORKLIST( $X$ ) to the method entry node
10:  PUSH( $X, \text{stack}$ )
11: end procedure
12: procedure DOANALYSIS
13:   $X \leftarrow \langle \text{main}, \text{BI} \rangle$ 
14:  INITCONTEXT( $X$ )
15:  while stack is not empty do
16:     $X \leftarrow \text{TOP}(\text{stack})$ 
17:     $w \leftarrow \text{WORKLIST}(X)$ 
18:     $n \leftarrow \text{REMOVE}(w)$  ▷ In Reverse Post Order
19:    Set IN( $X, n$ ) to the meet of all its predecessors
20:    Compute OUT( $X, n$ ) using the flow function of  $n$ 
21:    if OUT( $X, n$ ) has changed then
22:      if  $n$  is not the exit node then
23:        Add all successors of  $n$  to the worklist  $w$ 
24:      else if  $n$  is the exit node then
25:        Set EXITVALUE( $X$ ) to OUT( $X, n$ )
26:        POP(stack)
27:        for all  $(X', c) \rightarrow X$  in transitions do
28:          Add the call-site  $c$  to the worklist of  $X'$ 
29:          if  $X'$  is not on stack then
30:            PUSH( $X', \text{stack}$ )
31:          end if
32:        end for
33:      end if
34:    end if
35:  end while
36: end procedure
37: procedure PROCESSCALL( $X', c, m, v$ )
38:   $X \leftarrow \langle m, v \rangle$ 
39:  Add  $(X', c) \rightarrow X$  to transitions
40:  if  $X$  has not been previously created then
41:    INITCONTEXT( $X$ )
42:  end if
43:  return EXITVALUE( $X$ )
44: end procedure

```

**Figure 1.** The algorithm for performing value-based context-sensitive interprocedural analysis.

### Algorithm

Figure 1 provides the overall algorithm. Line 1 declares two globals: a stack of value contexts whose analysis is pending (each having its own work-list) and a transition table that maintains a mapping of calling context and call-site to the called context.

The procedure INITCONTEXT (lines 2-11) initializes a new context by first setting its exit value to the  $\top$  element of the lattice. Also, all nodes in the method body are initialized to the  $\top$  element, except the entry node, which is initialized to the given entry value. The work-list initially contains only the method entry point. Finally, the new context is pushed on the analysis stack.

The DOANALYSIS procedure (lines 12-37) first initializes a new value context for the *main* procedure and the analysis-specific boundary information (BI). Then, it performs data flow analysis using the traditional work-list based method for the context at the top of the stack (lines 15-23).

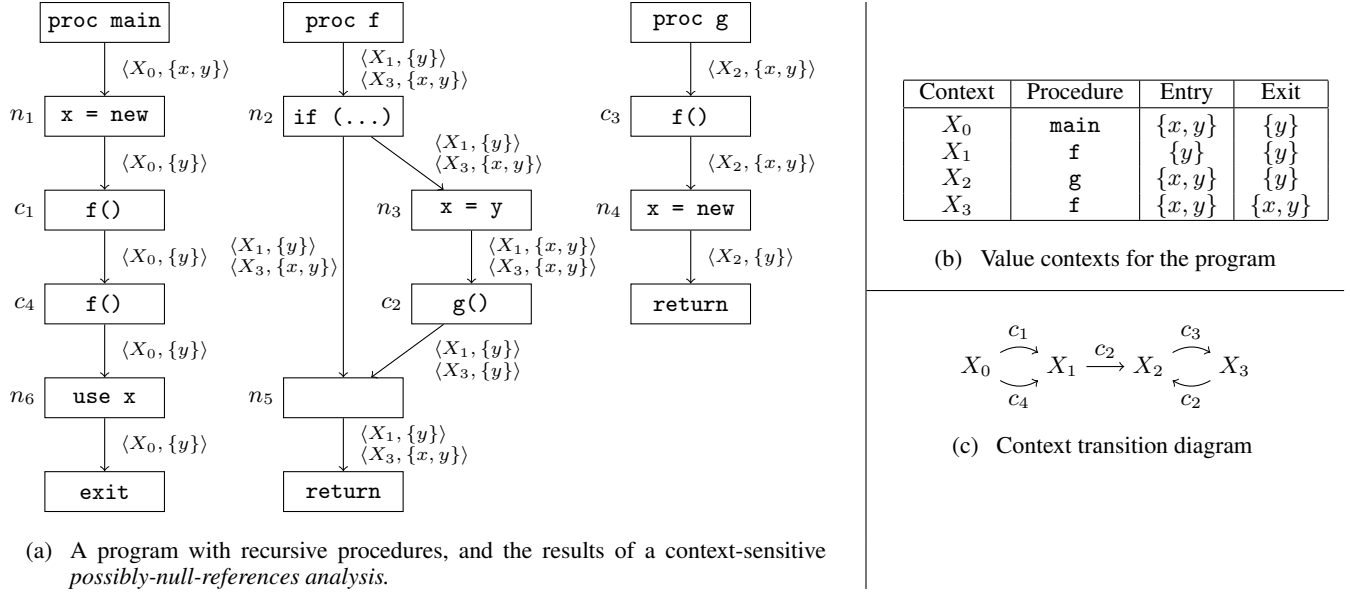


Figure 2. A motivating example.

A node is removed from the work-list and flow functions are processed in the usual fashion<sup>2</sup>. However, if the node contains a function call, then the procedure `PROCESSCALL` (lines 37-44) is invoked. In this procedure,  $X'$  is the value context being analyzed in the caller,  $c$  is the node containing the call,  $m$  is the called method, and  $v$  is the data flow value at the call-site. The value context at the entry of the called procedure is  $X = \langle m, v \rangle$ . `PROCESSCALL` records the transition of  $(X', c) \rightarrow X$  in the transition table and initializes the context  $X$  if this is the first time it has been encountered. `PROCESSCALL` then returns the exit flow value of  $X$ , which is  $\top$  for new contexts, and possibly some other value if  $X$  has been partially or fully analyzed before. Note that `DOANALYSIS` does not call `PROCESSCALL` explicitly. This is done by the flow function handler (not depicted here) when a call statement is encountered in a node. This handler would set the OUT value of the node using the value returned by `PROCESSCALL`.

Exit nodes of a procedure are processed (lines 24-33) for a context  $X$  as follows: `exitValue(X)` is recorded and  $X$  is popped off the stack. Then we consult the transition table to find all edges to  $X$  and add the call nodes of the callers to the work-lists of their respective contexts. When such a call node is removed from the work-list subsequently, the invocation of `PROCESSCALL` will return the exit flow value that we have just set on line 25.

### Example

Figure 2 illustrates our method for a program with mutually recursive procedures (`f` and `g`) accessing global variables  $x$  and  $y$ . We wish to discover possible null variable accesses in the program. Thus our data flow values are sets of variables that may be null at a given program point and the confluence operation is union. Although the flow functions in our example are distributive, the algorithm does not make any such assumption and does not place any constraints on flow functions. Also, it can be easily be used with minor modifications for backward data flow analyses.

The analysis starts at the entry of procedure `main` with the initial data flow value  $\{x, y\}$  because both variables are null in the

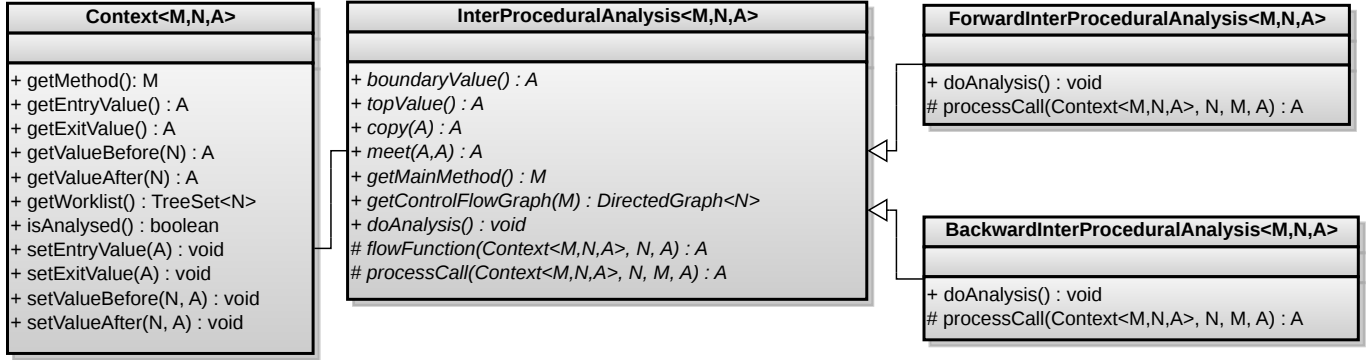
beginning. Thus the initial value context is  $X_0 = \langle \text{main}, \{x, y\} \rangle$  with exit value  $\top$ . The data flow values are recorded for each value context reaching a procedure (eg.  $\langle X_0, \{x, y\} \rangle$  just before node  $n_1$  in procedure `main`). Node  $n_1$  assigns a non-null value to  $x$  and our context sensitive data flow value at the exit of  $n_1$  becomes  $\langle X_0, \{y\} \rangle$ . At call-site  $c_1$ , a new context  $X_1 = \langle f, \{y\} \rangle$  is created, which is pushed on the stack and the transition  $(X_0, c_1) \rightarrow X_1$  is recorded. Figure 2 (b) lists all value contexts whereas Figure 2 (c) shows the associated transitions at call-sites.

Since  $X_1$  is on the top of the stack, analysis of procedure `f` begins for the value context  $X_1$  (i.e. data flow value  $\langle X_1, \{y\} \rangle$  at the entry of `f`). The assignment in  $n_3$  causes  $x$  to be considered as possibly null; the resulting data flow value is  $\langle X_1, \{x, y\} \rangle$ . Thus, the call site at  $c_2$  creates a new context  $X_2 = \langle g, \{x, y\} \rangle$  and a transition  $(X_1, c_2) \rightarrow X_2$ . In `g`, the first node is a call to `f`, but since there is no context for `f` with the data flow value  $\{x, y\}$ , we create a new context  $X_3 = \langle f, \{x, y\} \rangle$  and the transition  $(X_2, c_3) \rightarrow X_3$  and push  $X_3$  on the stack.

This initiates the analysis of `f` again but this time it is for the value context  $X_3$ . When we reach call-site  $c_2$ , we search for a value context for `g` with data flow value  $\{x, y\}$ . Since such a context already exists in the form of  $X_2$ , we only add the transition  $(X_3, c_2) \rightarrow X_2$ , and use its exit flow value (which is initially  $\top$ ). Since  $\top$  does not cause the OUT value at  $c_2$  to change, its successors are not added to the worklist. Instead, analysis proceeds along the branch  $n_2 \rightarrow n_5$  in `f` and reaches its exit node. This completes a partial analysis of `f` for the value context  $X_3$  and we set the exit value of  $X_3$  to  $\{x, y\}$ , and pop it off the stack.

From the transitions, we discover that the only caller of  $X_3$  is  $(X_2, c_3)$ , which is already on the stack. Hence  $c_3$  is added to the work-list of  $X_2$  and the analysis of `g` resumes with  $c_3$ . This requires us to examine a value context of the callee `f` with the data flow value  $\{x, y\}$  because the context sensitive data flow value at the IN of  $c_3$  is  $\langle X_2, \{x, y\} \rangle$ . We discover that the required value context is  $X_3$  with the exit flow  $\{x, y\}$  which is used in defining the OUT of  $c_3$  to  $\langle X_2, \{x, y\} \rangle$ . The analysis of `g` continues to its exit (killing  $x$  on the way). Now, as  $X_2$  is fully analyzed, its exit value is set and its callers  $(X_1, c_2)$  and  $(X_3, c_2)$  are added to their respective work-lists.  $X_2$  is popped from the stack, and now  $X_3$  is on top. In

<sup>2</sup>In order to ensure that all nodes are visited at least once, the domain of data flow values should be augmented by a dummy value that remains unmodified throughout the program.



**Figure 3.** The class diagram of our generic interprocedural analysis framework.

$X_3$ , we use the exit flow value of  $X_2$  at  $c_2$ , but as this does not change the value at  $n_5$ , which is  $\{x, y\}$ , we need not go further. The context is popped as its work-list becomes empty.  $X_1$  is next on the stack with node  $c_2$ , and its analysis continues to the exit node, finally propagating back to its caller ( $X_0, c_1$ ) in *main*. Thus the analysis of mutually recursive functions *f* and *g* has terminated without sacrificing precision.

At  $c_4$  in *main* we simply reuse *exitValue*( $X_1 = \langle f, \{y\} \rangle$ ) which is  $\{y\}$  and find that  $x$  is definitely not null at  $n_6$  and hence will not cause an exception. A context-insensitive analysis would have merged data flow values from all calls to *f* resulting in a conservative value of  $\{x, y\}$  to be propagated to *main*. A compiler would have then given a false warning that  $x$  may be null at  $n_6$ .

### 3. Implementation Framework

The implementation framework consists of a handful of core classes<sup>3</sup> as shown in Figure 3. The use of generic types makes the framework agnostic to any particular toolkit or IR. The classes are parameterized by three types: *M* represents the type of a method, *N* represents a node in the control flow graph and *A* is the type of data flow value used by the client analysis. The framework can be naturally instantiated for Soot using the type parameters *SootMethod* and *Unit* for *M* and *N* respectively.

The *Context* class encapsulates information about a value-context. Each context is created for a particular method. For forward flows, a context is constructed with an *entryValue*; the *exitValue* is initially  $\top$ . The converse is true for backward flows. The *getValueBefore* and *getValueAfter* methods retrieve data flow values just before and after a node respectively.

Users would extend *ForwardInterProceduralAnalysis* or *BackwardInterProceduralAnalysis*, both of which are subclasses of an abstract class *InterProceduralAnalysis*. The abstract methods *boundaryValue*, *topValue*, *copy* and *meet* are similar to their counterparts in Soot’s intraprocedural framework. These methods provide a hook for client analyses to express initial lattice values and basic operations on them. Additionally, methods *getMainMethod* and *getControlFlowGraph* are required to be implemented as this is an interprocedural framework. Currently we have borrowed the control flow graph representation from Soot’s graph toolkit, which is general and does not depend on Soot’s internals. The major functionality of the client analysis would be present in the *flowFunction* method which is parameterized by the context, the control flow graph node, and the data flow value.

The framework classes *ForwardInterProceduralAnalysis* and *BackwardInterProceduralAnalysis* implement two im-

portant methods described earlier in Figure 1: *doAnalysis* and *processCall*. The former is the entry point to the analysis and performs the management of the stack of contexts and their work-lists while the latter is used for processing a procedure call. The method *processCall* takes as parameters the value-context at the caller, the node in which the call-site appears, the procedure which is being called, and the data flow value at its entry (exit) for forward (backward) flows, and returns the data flow value at the exit (entry) of the target context, which is  $\top$  for a newly created context or the result of the previous partial or full analysis of the method body.

Note that the framework classes do not call *processCall* directly. It is called through the flow functions implemented in the client analysis. This design decision was deliberately taken because: (a) it allows for extensibility as the framework does not need to know whether *N* represents a statement or a basic block, and how to determine if there is a call inside it, and (b) the framework need not choose what methods are actually called from a given site—this is useful for analyses that build the call graph on-the-fly as discussed in Section 4.

### 4. The Role of Call Graphs

We initially developed this framework in order to implement heap reference analysis [6] using Soot, as this analysis cannot be encoded as an IFDS/IDE problem. However, even with our general framework, performing whole-program analysis turned out to be infeasible due to a large number of interprocedural paths arising from conservative assumptions for targets of virtual calls.

The SPARK engine [7] in Soot uses a flow and context insensitive pointer analysis on the whole program to build the call graph, thus making conservative assumptions for the targets of virtual calls in methods that are commonly used such as those in the Java library. For example, it is not uncommon to find call sites in library methods with 5 or more targets, most of which will not be traversed in a given context. Some call sites can even be found with more than 250 targets! This is common with calls to virtual methods defined in `java.lang.Object`, such as `hashCode()` or `equals()`.

When performing whole-program data flow analysis, the use of an imprecise call graph hampers both efficiency, due to an exponential blow-up of spurious paths, and precision, due to the meet over paths that are actually interprocedurally invalid, thereby diminishing the gains from context-sensitivity.

Soot provides a context-sensitive call graph builder called PADDLE [8], but this framework can only perform *k*-limited call-site or object-sensitive analysis, and that too in a flow-insensitive manner. We were unable to use PADDLE with our framework directly because at the moment it not clear to us how the *k*-suffix contexts of PADDLE would map to our value-contexts.

<sup>3</sup> Some internally used methods have been omitted for brevity.

Benchmark		Time	Methods ( $M$ )		Contexts ( $X$ )		$X/M$		Clean	
			Total	App.	Total	App.	Total	App.	Total	App.
SPEC JVM98	compress	1.15s	367	54	1,550	70	4.22	1.30	50	47
	jess	140.8s	690	328	17,280	9,397	25.04	28.65	34	30
	db	2.19s	420	56	2,456	159	5.85	2.84	62	46
	mpegaudio	4.51s	565	245	2,397	705	4.24	2.88	50	47
	jack	89.33s	721	288	7,534	2,548	10.45	8.85	273	270
DaCapo 2006	antlr	697.4s	1,406	798	30,043	21,599	21.37	27.07	769	727
	chart	242.3s	1,799	598	16,880	4,326	9.38	7.23	458	423

**Table 1.** Results of points-to analysis using our framework. “App.” refers to data for application classes only.

### Call Graph Construction using Points-To Analysis

We have implemented a flow and context sensitive points-to analysis using our interprocedural framework to build a call graph on-the-fly. This analysis is both a demonstration of the use of our framework as well as a proposed solution for better call graphs intended for use by other interprocedural analyses.

The data flow value used in our analysis is a points-to graph in which nodes are allocation sites of objects. We maintain two types of edges:  $x \rightarrow m$  indicates that the root variable  $x$  may point to objects allocated at site  $m$ , and  $m.f \rightarrow n$  indicates that objects allocated at site  $m$  may reference objects allocated at site  $n$  along the field  $f$ . Flow functions add or remove edges when processing assignment statements involving reference variables. Nodes that become unreachable from root variables are removed. Type consistency is maintained by propagating only valid casts.

The points-to graphs at each statement only maintain objects reachable from variables that are local to the method containing the statement. At call statements, we simulate assignment of arguments to locals of the called method, as well as the assignment of returned values to a local of the caller method. For static fields (and objects reachable from them) we maintain a global flow-insensitive points-to graph. For statements involving static loads/stores we operate on a temporary union of local and global graphs. The call graph is constructed on-the-fly by resolving virtual method targets using type information of receiver objects.

Points-to information cannot be precise for objects returned by native methods, and for objects shared between multiple threads (as our analysis is flow-sensitive). Thus, we introduce the concept of a *summary node*, which represents statically unpredictable points-to information and is denoted by the symbol  $\perp$ . For soundness, we must conservatively propagate this effect to variables and fields that involve assignments to summary nodes. The rules for summarization along different types of assignment statements are as follows:

Statement	Rule used in the flow function
$x = y$	If $y \rightarrow \perp$ , then set $x \rightarrow \perp$
$x.f = y$	If $y \rightarrow \perp$ , then $\forall o : x \rightarrow o$ , set $o.f \rightarrow \perp$
$x = y.f$	If $y \rightarrow \perp$ or $\exists o : y \rightarrow o$ and $o.f \rightarrow \perp$ , then set $x \rightarrow \perp$
$x = p(a_1, a_2, \dots)$	If $p$ is unknown, then set $x \rightarrow \perp$ , and $\forall o : a_i \rightarrow o, \forall f \in \text{fields}(o)$ set $o.f \rightarrow \perp$

The last rule is drastically conservative; for soundness we must assume that a call to an unknown procedure may modify the fields of arguments in any manner, and return any object. An important discussion would be on what constitutes an *unknown* procedure. Native methods primarily fall into this category. In addition, if  $p$  is a virtual method invoked on a reference variable  $y$  and if  $y \rightarrow \perp$ , then we cannot determine precisely what the target for  $p$  will be. Hence, we consider this call site as a *default* site, and do not enter the procedure, assuming worst-case behaviour for its arguments and returned values. A client analysis using our framework can

choose to do one of two things when encountering a *default* call site: (1) assume worst case behaviour for its arguments<sup>4</sup> and carry on to the next statement, or (2) fall-back onto Soot’s default call graph and follow the targets it gives.

A related approach partitions a call graph into calls from application classes and library classes [2]. Our call graph is partitioned into call sites that can be statically resolved to one or more valid targets, and those that cannot due to statically unpredictable types.

### Experimental Results

Table 1 lists the results of points-to analysis performed on seven benchmarks. The experiments were carried out on an Intel Core i7-960 with 19.6 GB of RAM running Ubuntu 12.04 (64-bit) and JDK version 1.6.0.27. Our single-threaded analysis used only one core.

The first two columns contain the names of the benchmarks; five of which are the single-threaded programs from the SPEC JVM98 suite [1], while the last two<sup>5</sup> are from the DaCapo suite [3] version 2006-10-MR2. The third column contains the time required to perform our analysis, which ranged from a few seconds to a few minutes. The fourth and fifth columns contain the number of methods analyzed (total and application methods respectively). The next two columns contain the number of value-contexts created, with the average number of contexts per method in the subsequent two columns. It can be seen that the number of distinct data flow values reaching a method is not very large in practice.

The use of *default* sites in our call graph has two consequences: (1) the total number of analyzed methods may be less than the total number of reachable methods and (2) methods reachable from *default* call sites (computed using SPARK’s call graph) cannot be soundly optimized by a client analysis that jumps over these sites. The last column lists the number of *clean* methods which are not reachable from *default* sites and hence can be soundly optimized. In all but two cases, the majority of application methods are clean.

In order to highlight the benefits of using the resulting call graph, just listing the number of edges or call-sites alone is not appropriate, as our call graph is context-sensitive. We have thus computed the number of distinct call chains (i.e. paths in the call graph) starting from the entry point, which are listed in Table 2. As the total number of call chains is possibly infinite (due to recursion), we have counted chains up to a maximum length of  $k$ , for  $1 \leq k \leq 10$ . For each benchmark, we have counted the call chains using call graphs constructed by our Flow and Context sensitive Pointer Analysis (FCPA) as well as SPARK<sup>6</sup>, and noted the difference as percentage savings ( $\Delta\%$ ) from using our context-sensitive call graph.

<sup>4</sup>For example, in liveness analysis, a worst-case assumption can be that all arguments and objects reachable from them are live.

<sup>5</sup>As stub classes are needed to simulate DaCapo’s reflective boot process, our analysis is inappropriate for other benchmarks in this suite because it ignores paths with method invocations on null pointers.

<sup>6</sup>The option `implicit-entry` was set to `false` for SPARK.

Depth $k =$		1	2	3	4	5	6	7	8	9	10
compress	FCPA	2	5	7	20	55	263	614	2,225	21,138	202,071
	SPARK	2	5	9	22	57	273	1,237	23,426	545,836	12,052,089
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>22.2</b>	<b>9.09</b>	<b>3.51</b>	<b>3.66</b>	<b>50.36</b>	<b>90.50</b>	<b>96.13</b>	<b>98.32</b>
jess	FCPA	2	5	7	30	127	470	4,932	75,112	970,044	15,052,927
	SPARK	2	5	9	32	149	924	24,224	367,690	8,591,000	196,801,775
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>22.2</b>	<b>6.25</b>	<b>14.77</b>	<b>49.13</b>	<b>79.64</b>	<b>79.57</b>	<b>88.71</b>	<b>92.35</b>
db	FCPA	2	5	11	46	258	1,791	21,426	215,465	2,687,625	42,842,761
	SPARK	2	5	13	48	443	4,726	71,907	860,851	13,231,026	245,964,733
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>15.4</b>	<b>4.17</b>	<b>41.76</b>	<b>62.10</b>	<b>70.20</b>	<b>74.97</b>	<b>79.69</b>	<b>82.58</b>
mpegaudio	FCPA	2	14	42	113	804	11,286	129,807	1,772,945	27,959,747	496,420,128
	SPARK	2	16	46	118	834	15,844	250,096	4,453,608	87,096,135	1,811,902,298
	$\Delta\%$	<b>0</b>	<b>12</b>	<b>8.7</b>	<b>4.24</b>	<b>3.60</b>	<b>28.77</b>	<b>48.10</b>	<b>60.19</b>	<b>67.90</b>	<b>72.60</b>
jack	FCPA	2	18	106	1,560	22,652	235,948	2,897,687	45,480,593	835,791,756	17,285,586,592
	SPARK	2	18	106	1,577	27,201	356,867	5,583,858	104,211,833	2,136,873,586	46,356,206,503
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>0</b>	<b>1.08</b>	<b>16.72</b>	<b>33.88</b>	<b>48.11</b>	<b>56.36</b>	<b>60.89</b>	<b>62.71</b>
antlr	FCPA	6	24	202	560	1,651	4,669	18,953	110,228	975,090	11,935,918
	SPARK	6	24	206	569	1,669	9,337	107,012	1,669,247	27,670,645	468,973,725
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>1.9</b>	<b>1.58</b>	<b>1.08</b>	<b>49.99</b>	<b>82.29</b>	<b>93.40</b>	<b>96.48</b>	<b>97.45</b>
chart	FCPA	6	24	217	696	2,109	9,778	45,010	517,682	7,796,424	164,476,462
	SPARK	6	24	219	714	2,199	20,171	306,396	7,676,266	192,839,216	4,996,310,985
	$\Delta\%$	<b>0</b>	<b>0</b>	<b>0.9</b>	<b>2.52</b>	<b>4.09</b>	<b>51.52</b>	<b>85.31</b>	<b>93.26</b>	<b>95.96</b>	<b>96.71</b>

**Table 2.** Number of call chains up to depth  $k$  for various benchmarks using SPARK and FCPA (Flow and Context-sensitive Pointer Analysis).

The savings can be clearly observed for  $k > 5$ . For  $k = 10$ , SPARK's call graph contains more than 96% spurious paths for three of the benchmarks, and 62-92% for the remaining. The gap only widens for larger values of  $k$  (for which the number of chains was too large to compute in some cases). Our context-sensitive call graph avoids these spurious paths and allows for efficient and precise interprocedural data flow analysis using our framework.

## 5. Conclusion and Future Work

We have presented a framework for performing value-based context-sensitive inter-procedural analysis in Soot. This framework does not require distributivity of flow functions and is thus applicable to a large class of analyses including those that cannot be encoded as IFDS/IDE problems.

In order to deal with the difficulties in whole-program analysis performed over an imprecise call graph, we have proposed a solution by constructing call graphs that are computed on-the-fly while performing a flow and context sensitive points-to analysis. This analysis also demonstrated a sample use of our framework and showed that it is practical to use data flow values as contexts because the number of distinct data flow values at call sites to a method are usually not very large.

Our implementation of the interprocedural framework is mature and ready for release. However, our points-to analysis implementation is experimental and makes heavy use of HashMaps and HashSets, thus running out of memory for very large programs. We would like to improve this implementation by using bit-vectors or maybe even BDDs for compact representation of points-to sets.

The precision of our call graphs is still limited in the presence of *default* sites, which are prominent due to the liberal use of *summary* nodes in the points-to graphs. We would like to reduce the number of *summary* nodes by simulating some commonly used native methods in the Java library, and also by preparing a summary of initialized static fields of library classes. Our hope is that these extensions would enable our call graph to be fully complete, thereby enabling users to precisely perform whole-program analysis and optimization for all application methods in an efficient manner. We believe that improving the precision of program analysis actually helps to improve its efficiency, rather than hamper it.

## References

- [1] <http://www.spec.org/jvm98>. Accessed: April 3, 2013.
- [2] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European conference on Object-Oriented Programming, ECOOP'12*, 2012.
- [3] S. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2006.
- [4] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP '12*, 2012.
- [5] U. P. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08*, 2008.
- [6] U. P. Khedker, A. Sanyal, and A. Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1), Nov. 2007.
- [7] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction, CC'03*, 2003.
- [8] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), Oct. 2008.
- [9] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '95*, 1995.
- [10] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), Oct. 1996.
- [11] M. Sharir and A. Pnueli. Two approaches to interprocedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.
- [12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, CASCON '99*, 1999.